Конспект лекции
Архитектура JVM

Цель и задачи лекции

Цель – изучить архитектуру JVM.

Задачи:

1. Изучить архитектуру виртуальной машины Java
2. Изучить как происходит сборка мусора в Java
3. Понять, как работает JIT-компилятор

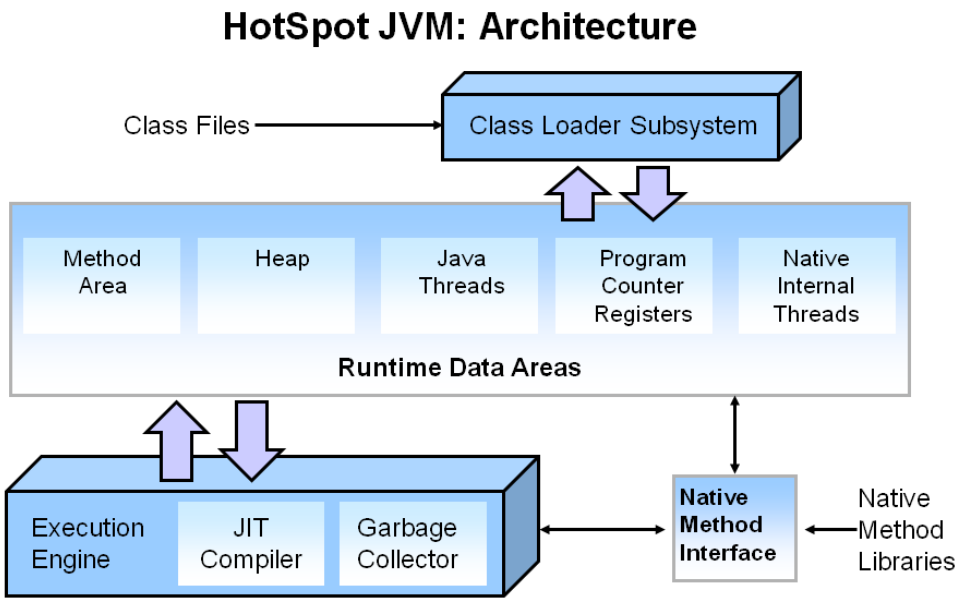
План занятия

1. Архитектура JVM
2. Сборка мусора в Java
3. JIT-компиляция

Архитектура JVM

Java Virtual Machine — виртуальная машина Java — основная часть исполняющей системы Java, так называемой Java Runtime Environment (JRE). Виртуальная машина Java исполняет байт-код Java, предварительно созданный из исходного текста Java-программы компилятором Java (javac). JVM может также использоваться для выполнения программ, написанных на других языках программирования. Например, исходный код на языке Ada может быть откомпилирован в байт-код Java, который затем может выполнится с помощью JVM.

JVM является ключевым компонентом платформы Java. Так как виртуальные машины Java доступны для многих аппаратных и программных платформ, Java может рассматриваться и как связующее программное обеспечение, и как самостоятельная платформа. Использование одного байт-кода для многих платформ позволяет описать Java как «скомпилировано однажды, запускается везде» (compile once, run anywhere).

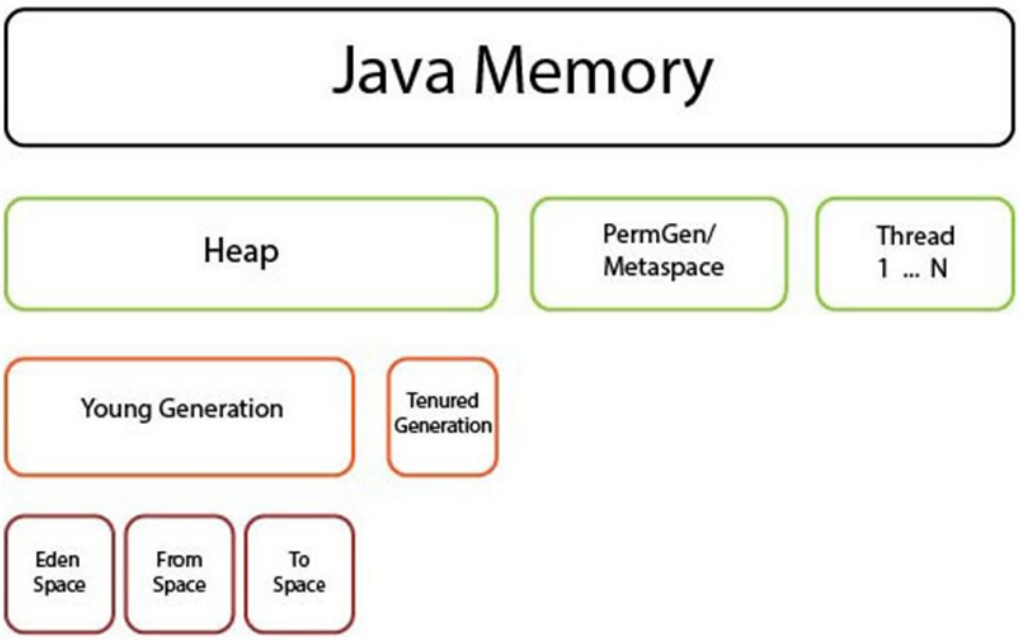


Структура памяти Java

Память Java состоит из трех компонентов: heap, stack и metaspace (который ранее назывался permgen).

- Metaspace, где Java держит свои программы неизменных информация как определения классов.
- Пространство кучи (heap), где Java держит переменные.
- Пространство стека (stack), где Java сохраняет функции исполнения и ссылки на переменные.

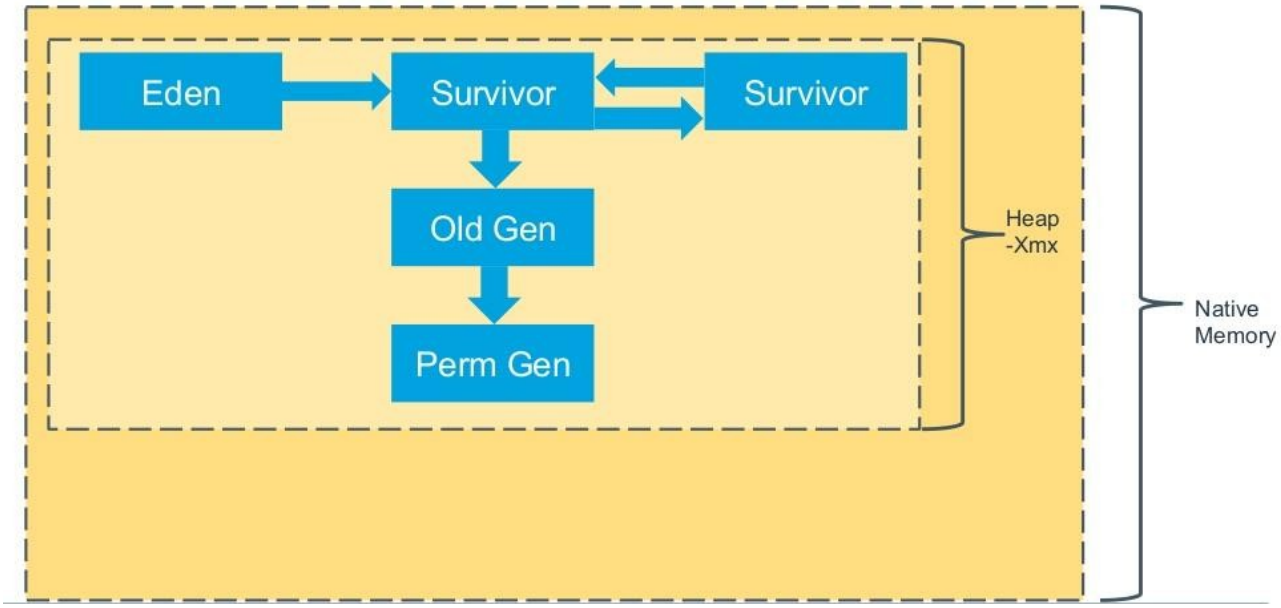
До Java 8, metaspace был известен как permgen.



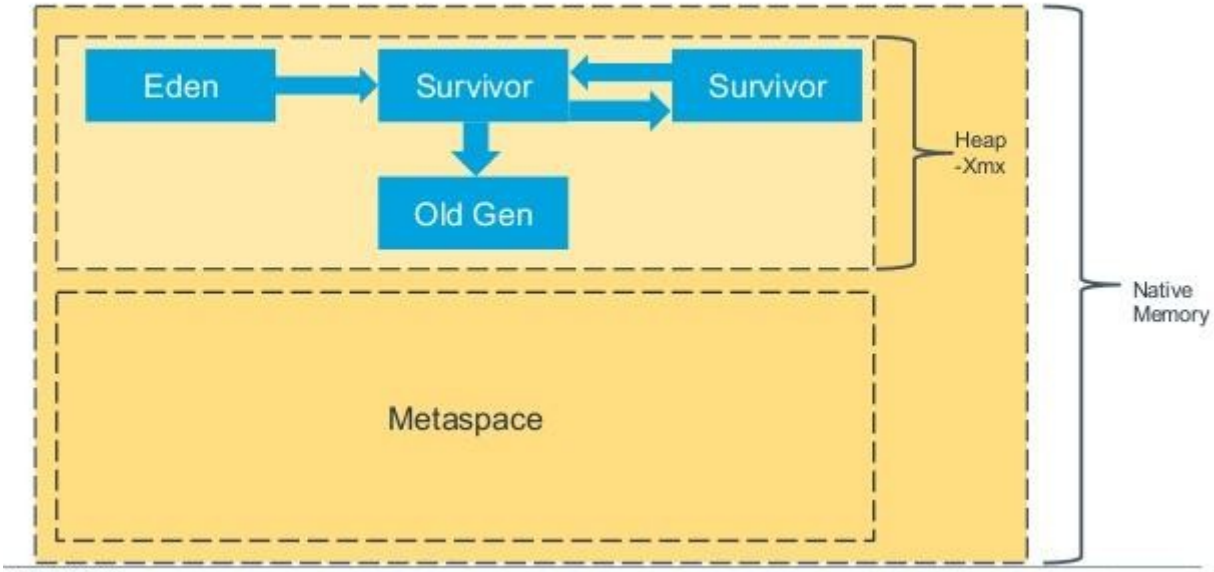
Память процесса различается на heap (куча) и non-heap (стек) память, и состоит из 5 областей (memory pools, memory spaces):

- Eden Space (heap) – в этой области выделяется память под все создаваемые из программы объекты. Большая часть объектов живет недолго (итераторы, временные объекты, используемые внутри методов и т.п.), и удаляются при выполнении сборок мусора это области памяти, не перемещаются в другие области памяти. Когда данная область заполняется (т.е. количество выделенной памяти в этой области превышает некоторый заданный процент), GC выполняет быструю (minor collection) сборку мусора. По сравнению с полной сборкой мусора она занимает мало времени, и затрагивает только эту область памяти — очищает от устаревших объектов Eden Space и перемещает выжившие объекты в следующую область.
- Survivor Space (heap) – сюда перемещаются объекты из предыдущей, после того, как они пережили хотя бы одну сборку мусора. Время от времени долгоживущие объекты из этой области перемещаются в Tenured Space.
- Tenured (Old) Generation (heap) — Здесь скапливаются долгоживущие объекты (крупные высокоуровневые объекты, синглтоны, менеджеры ресурсов и проч.). Когда заполняется эта область, выполняется полная сборка мусора (full, major collection), которая обрабатывает все созданные JVM объекты.
- Metaspace (non-heap) – Здесь хранится метаинформация, используемая JVM (используемые классы, методы и т.п.). В частности
- Code Cache (non-heap) — эта область используется JVM, когда включена JIT-компиляция, в ней кешируется скомпилированный платформенно — зависимый код.

Модель памяти в Java 7 представлена на изображении ниже



Модель памяти в Java 8 представлена на изображении ниже



Рассмотрим код, который вызывает переполнение памяти:

```
public class OutOfMemoryError {

    public static void main(String args[]) {
        OutOfMemoryError ome = new OutOfMemoryError();
        ome.generateMyIntArray(1,50);
    }

    public void generateMyIntArray(int start, int end) {
        int multiplier = 100;
        for(int i = 1; i < end; i++) {
            System.out.println("Round " + i + " Free Memory: " +
Runtime.getRuntime().freeMemory());
            int[] myIntList = new int[multiplier];
            for(int j= i; j > 1; j--){
                myIntList[j] = i;
            }
            multiplier = multiplier * 10;
        }
    }
}
```

При компиляции и выполнении программы будет выведено следующее:

```
Round 1 Free Memory: 254741016
Round 2 Free Memory: 254741016
Round 3 Free Memory: 254741016
Round 4 Free Memory: 254741016
Round 5 Free Memory: 254741016
Round 6 Free Memory: 250741000
Round 7 Free Memory: 210740984
Round 8 Free Memory: 210772712
Exception in thread “main” java.lang.OutOfMemoryError: Java heap space  at
OutOfMemoryError.generateMyIntArray(OutOfMemoryError.java:16) at
com.codenuclear.b.OutOfMemoryError.main(OutOfMemoryError.java:8)
```

Ошибка OutOfMemoryError, как следует из названия, возникает, когда сборщик мусора достиг своего предела накладных расходов. Это означает, что он

работает все время, но очень медленно собирает объекты. Рассмотрим следующий пример:

```
public class GCOverheadTest {
    public static void main(String args[]) throws Exception {
        Map<Object,Object> map = System.getProperties();
        Random r = new Random();
        while (true) {
            map.put(r.nextInt(), "GC Overhead Limit Test");
        }
    }
}
```

Для анализа результатов приведенного кода нужно запустить программу с параметрами:

java -Xmx100m -XX:+UseParallelGC GCOverheadTest

В результате мы получим ошибку:

```
Exception in thread “main” java.lang.OutOfMemoryError: GC overhead limit exceeded
      at java.util.Hashtable.addEntry(Unknown Source)
      at java.util.Hashtable.put(Unknown Source)
      at GCOverheadTest.main(GCOverheadTest.java:10)
```

Чтобы повысить производительность сборщика мусора, JVM настроена на обнаружение того, что в процессе сбора мусора процесс Java по умолчанию восстанавливает менее 2 процентов кучи, затрачивая более 98 процентов своего времени обработки. Когда это происходит, он выдает ошибку превышения предельного значения GC. Эта проверка необходима, потому что, если GC собирается восстановить только 2 процента памяти, не будет много места для новых объектов. Следовательно, JVM будет запускать процесс GC снова и снова, только чтобы претендовать на это крошечное пространство. Эта операция будет использовать ресурсы ЦП полностью, и приложение больше не будет отвечать на запросы пользователей. Следовательно, эта проверка необходима. Однако мы можем отключить эту проверку, добавив флаг -XX: -UseGCOverheadLimit к параметру VM.

Сборка мусора в Java

Java HotSpot VM предоставляет разработчикам на выбор четыре различных сборщика мусора:

- Serial (последовательный) — самый простой вариант для приложений с небольшим объемом данных и не требовательных к задержкам. Данный сборщик мусора используется довольно редко, однако на слабых компьютерах может быть выбран виртуальной машиной в качестве сборщика по умолчанию.
- Parallel (параллельный) — наследует подходы к сборке от последовательного сборщика, но добавляет параллелизм в некоторые операции, а также возможности по автоматической подстройке под требуемые параметры производительности.

- Concurrent Mark Sweep (CMS) — нацелен на снижение максимальных задержек путем выполнения части работ по сборке мусора параллельно с основными потоками приложения. Подходит для работы с относительно большими объемами данных в памяти.
- Garbage-First (G1) — создан для постепенной замены CMS, особенно в серверных приложениях, работающих на многопроцессорных серверах и оперирующих большими объемами данных.

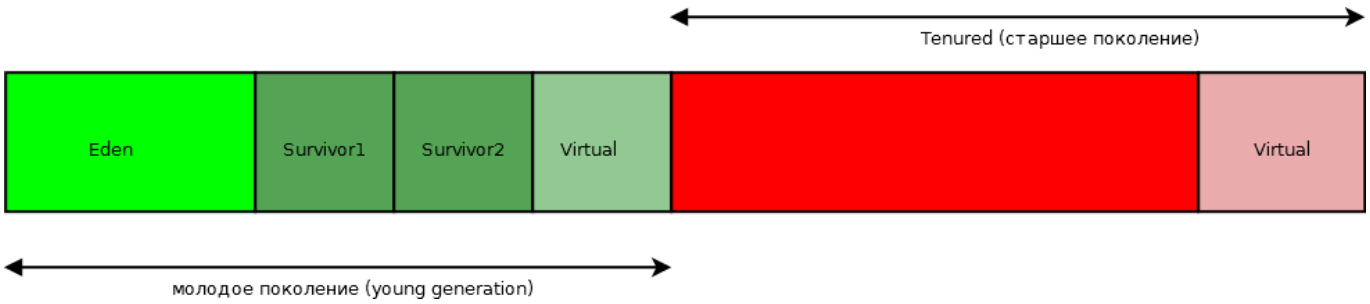
Serial GC

Serial garbage collector – последовательный сборщик мусора. Последовательным он называется, потому что при его работе выполнение программы приостанавливается.

Для начала нужно определить, какой объект нам уже можно удалять. Объект считается мусором (то есть его можно удалить) тогда, когда его больше нельзя достичь ни из какой точки работающей программы. Самый простой алгоритм, который можно сделать в лоб, — это просто пройтись по всем объектам, которые можно достичь, тогда те объекты, которые мы не посетили будут мусором, и их можно будет удалить.

Однако последовательный сборщик мусора, описываемый в этой статье, использует поколения объектов, для уменьшения количества обходимых за один раз объектов. Идея состоит в том, что большая часть объектов существует очень короткое время (различные итераторы, временные объекты для передачи параметров и др.), и лишь небольшая кучка объектов существует достаточно долго.

В соответствии с описанным выше алгоритм сборки serial garbage collector использует два поколения: молодое и старое. Молодое поколение делится на: Eden, Survivor1 и Survivor2, как на изображении:



Если дословно переводить с английского, то Eden — это Эдем, райский сад из Библии, место первоначального обитания людей; survivor — уцелевший, спасшийся; tenured — штатный.

При инициализации большая часть адресного пространства резервируется виртуально, но не выделяется физически (на изображении это области Virtual).

Новые объекты почти всегда создаются в Eden, исключение могут составлять большие объекты, которые накладно постоянно перемещать между областями survivor. Большая часть объектов там же и уничтожается.

Одна из областей survivor всегда пуста. При первой сборке мусора в молодом поколении выжившие объекты перемещаются в один из регионов survivor. При второй сборке мусора в молодом поколении смотрятся объекты в Eden и в заполненном survivor, а все выжившие объекты переносятся в свободный survivor и т. д.

Объекты переносятся между областями survivor до тех пор, пока они не будут достаточно стары, чтобы их можно было переместить в tenured. Когда область tenured заполнится, то включается полная сборка мусора, которая затрагивает объекты из обоих поколений, и которая длится дольше, чем сборка только в молодом поколении.

Обобщенно, алгоритм работы Serial GC примерно следующий:

1. Когда нет места в Eden, запускается GC, живые объекты копируются в S1. Вся область Eden очищается. S1 и S2 меняются местами.
2. При последующих циклах в S1 будут записаны живые объекты как из Eden, так и из S2. После нескольких циклов обмена S1 и S2 или заполнения области S2, объекты, которые живут достаточно долго перемещаются в Old Greneration. Следует сказать, что не всегда объекты при создании аллоцируются в Eden. Если объект слишком велик, он сразу идет в Old Generation.
3. Когда после очередной сборки мусора места не хватает уже в New Generation, то запускается сбор мусора в Old Generation (наряду со сборкой New Generation). В Щld Generation объекты уплотняются (алгоритм Mark-Sweep-Compact).
4. Если после полной сборки мусора места не хватает, то возникнет ошибка Java.lang.OutOfMemoryError.

Но во время работы может запрашиваться увеличение памяти и Heap может увеличиваться. Как правило, Old Generation занимает 2/3 объема Heap.

Эффективность алгоритма сборки мусора считается по параметру STW (Stop The World) - время, когда все процессы кроме GC останавливаются. Serial в этом смысле не слишком эффективен, т.к. делает свою работу не торопясь, в одном потоке.

Parralell GC

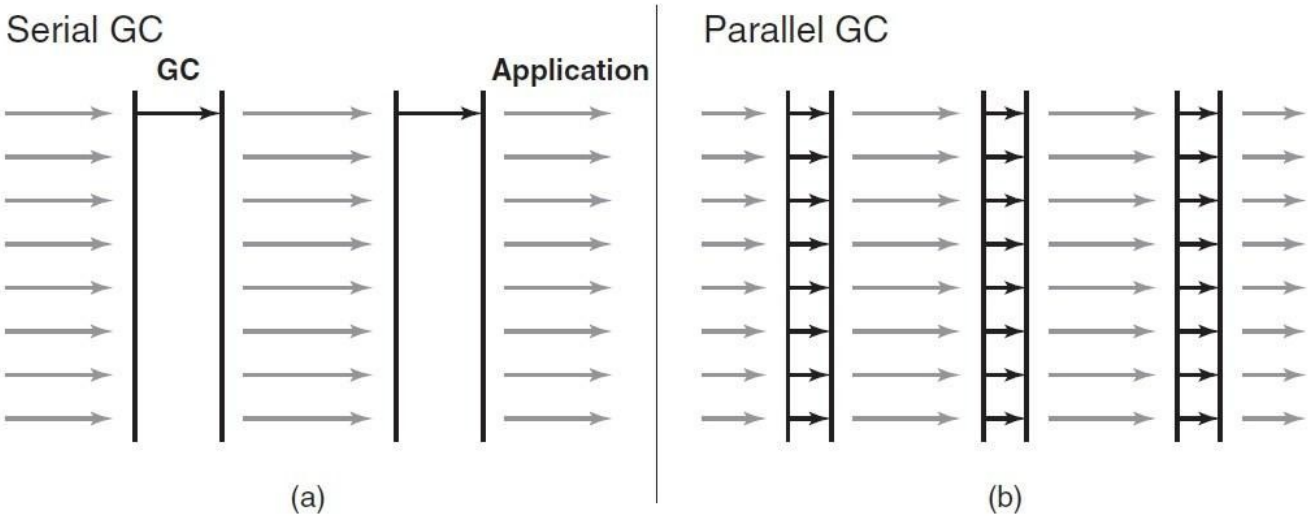
Parallel GC (параллельный сборщик) - перенимает идеи Serial GC добавляя в них параллелизм. JVM может автоматически задействовать его если на вашем компьютере несколько ядер.

Включить параллельный сборщик мусора можно с помощью аргумента -XX:+UseParallelGC, переданного в качестве параметра командной строки виртуальной машины Java. Параллельный сборщик мусора выбирается по умолчанию для серверных машин.

По умолчанию параллельно основным потокам программы происходит сборка мусора как из молодого поколения, так и полная.

Параллельный сборщик мусора особенно полезен для систем с несколькими физическими процессорами. Для машин с N физическими процессорами Parallel GC создаёт N потоков для случая N<8 и целую часть от дроби 5/8 при N>=8.

Количество создаваемых потоков можно задать с помощью командной строки -XX:ParallelGCThreads=<N>.



Так как при сборке мусора работают несколько потоков, то при перемещении объектов из молодого поколения в старшее возможна фрагментация старшего поколения, так как каждому потоку выделяется своя, изолированная от остальных область в tenured.

При тюнинге parallel GC можно указать не размеры поколений и другие низкоуровневые параметры, а более высокоуровневые настройки: максимальную паузу для сборки мусора, throughput (производительность) и footprint (размер кучи).

Максимальная пауза для сборки мусора устанавливается с помощью параметра XX:MaxGCPauseMillis=<N> в миллисекундах. Причём указанное значение не всегда будет успешно выдерживаться, но сборщик мусора будет пытаться подстраивать размер кучи и другие параметры так, чтобы сборка мусора была меньше указанного здесь значения.

Throughput или производительность устанавливается с помощью параметра -XX:GCTimeRatio=<N>, где указывается отношение времени затраченного на сборку мусора к остальному времени работы приложения. Parallel GC будет стараться придерживаться этого целевого значения производительности.

Footprint или максимальный размер кучи устанавливается с помощью параметра -Xmx<N>, причём сборщик будет пытаться выдерживать как можно меньший размер кучи при достижении остальных целевых параметров.

Цели рассматриваются в следующем приоритете:

- Максимальная пауза при сборке мусора
- Производительность
- Минимальный размер кучи.

Если больше 98 % времени тратится на сборку мусора и удалось очистить меньше 2 % кучи, то параллельный сборщик мусора бросает исключение OutOfMemoryError.

Размеры поколений при необходимости увеличиваются и уменьшаются. По умолчанию увеличение идёт на 20 %, а уменьшение на 5 %. Эти значения настраиваются с помощью параметров командной строки. Для шага увеличения размера молодого поколения используйте параметр -XX:YoungGenerationSizeIncrement=<Y> (в процентах), для шага увеличения размера старшего поколения используйте -XX:TenuredGenerationSizeIncrement=<T> (тоже в процентах). Для шага уменьшения размеров поколения используется -XX:AdaptiveSizeDecrementScaleFactor=<D>, где процент уменьшения размера поколения вычисляется по формуле X/D, (X — процент увеличения размера поколения, D — значение параметра -XX:AdaptiveSizeDecrementScaleFactor).

В JDK 9 сборщиком мусора по умолчанию стал G1, в то время как раньше базовым сборщиком был Parallel GC, который мог собирать мусор в нескольких потоках. Теперь это сможет и G1, раньше он делал это в одном потоке, что иногда вызывало сложности. Разработчики могут настраивать количество потоков с помощью параметра -XX:ParallelGCThreads.

При включении параллельного сборщика мусора используются те же самые подходы что и в GC, та же разбивка памяти на Eden, S-0, S-1, Tunered, те же алгоритмы сборки minor cycle и major cycle, но во первых сборка мусора происходит в параллельном режиме, во вторых, добавляется “интеллектуальность” для оптимизации производительности о которой мы поговорим ниже.

Для параллельной обработки объектов, действует немного изменённый механизм переноса. Так например каждый поток GC получает свой участок памяти куда он будет копировать выжившие объекты чтобы не мешать другим потокам. Это дает ускорение, но немного дефрагментирует память.

Интеллектуальной составляющей как раз является то, что производительность сборки мусора (так и самого приложения) можно настраивать так, чтобы она удовлетворяла потребности. Вы можете указать устраивающие вас параметры производительности — максимальное время сборки и/или пропускную способность — и сборщик будет изо всех сил стараться не превышать заданные пороги. Для этого он будет использовать статистику уже прошедших сборок мусора и исходя из нее планировать параметры дальнейших сборок: варьировать размеры поколений, менять пропорции регионов.

Concurrent Mark Sweep

Concurrent Mark Sweep (CMS) использует параллельный mark-copy алгоритм в молодом поколении и, в основном, параллельный mark-sweep алгоритм в старом поколении.

Эти переключатели переопределяют стандартные настройки сборки мусора, конфигурируя сборщик мусора CMS с N параллельных потоков для сборки. Такой сборщик будет выполнять максимальное количество работы по сборке, реализуемое при параллельной обработке в данной системе.

Существует три ключевых фактора алгоритма отслеживания и очистки, которые важны в данном случае:

- неизбежно некоторое количество пауз, когда приостанавливаются все работающие потоки;
- подсистема сборки мусора ни в коем случае не должна собирать «живые» объекты — в противном случае мы рискуем аварийным завершением виртуальной машины Java или даже чем-нибудь похуже;
- вы можете гарантированно собрать весь мусор лишь в том случае, если на время сборки все потоки приложения будут остановлены.

При работе CMS активно использует именно последний пункт. Он делает две очень краткие паузы с полной остановкой всех потоков, а также работает параллельно с другими потоками приложения в оставшуюся часть цикла по сборке мусора. Таким образом, CMS избегает получения «ложноотрицательных» результатов, но, так как в описанной ситуации возникают условия гонки, может упустить часть мусора (такой упущенный мусор будет собран в следующем цикле).

Кроме того, CMS при работе приходится вести более сложный учет того, что является и что не является мусором. Приходится пойти на эти дополнительные издержки, чтобы работа могла протекать в основном без остановки потоков приложения. Обычно такая работа лучше выполняется на машинах с большим количеством ядер, где возможны более частые и краткие паузы.

Алгоритм CMS для Old Generation по-прежнему пытается работать с потоками приложения и, таким образом, вступает в конкуренцию за процессорное время. По умолчанию этот алгоритм GC использует количество потоков, равное 1/4 от количества физических ядер ЦП.

G1 GC

В Java 8 сборщиком мусора по умолчанию был Parallel GC, в Java 9 это изменилось. Ему на смену пришёл G1 Garbage Collector, который обеспечивает минимальное время stop-the-world (время когда сборщик мусора останавливает работу приложения).

Сборщик мусора Garbage-First (G1) нацелен на многопроцессорные системы с большим количеством памяти. Он старается с высокой точностью достичь заданной цели по времени остановки выполнения основной программы, но при этом добиться высокой пропускной способности при

небольшой потребности в настройке. G1 нацелен на баланс между задержками и пропускной способностью.

В Java 9 сборщик мусора G1 используется по умолчанию, но вы также можете явно включить с помощью опции -XX:+UseG1GC.

Также, как и остальные сборщики мусора G1 разбивает кучу на молодое и старое поколения. Сборка мусора происходит по большей части в молодом поколении, чтобы увеличить пропускную способность сборки, сборка в старом поколении происходит гораздо реже.

Некоторые операции всегда выполняется в stop-the-world, чтобы увеличить пропускную способность, но некоторые операции, которые занимают большое время, например работающие со всей кучей, осуществляются параллельно с работой основного приложения. Для уменьшения времени stop-the-world сборщик мусора G1 осуществляет сборку инкрементно и параллельно с работой основного приложения. Сборщик мусора G1 собирает информацию о предыдущих сборках мусора, чтобы добиться более точного соблюдения времени stop-the-world. Например, он собирает мусор в первую очередь в тех местах, которые заполнены больше всего.

G1 переносит живые объекты из выбранных областей памяти в новые области, тем самым упаковывая их рядом для более эффективного использования памяти. После переноса выживших объектов память, которая была занята ими до этого процесса используется для новых объектов приложения.

Garbage-First collector — это НЕ сборщик мусора реального времени. Он пытается достичь заданного времени по stop-the-world, но это время не выдерживается для конкретной одной остановки приложения.

Сборщик G1 разбивает кучу на регионы одинакового размера. Каждый регион может быть либо свободен, либо содержать объекты из молодого поколения, либо содержать объекты из старого поколения. Регионов достаточно много, может быть множество регионов с молодым поколением и множество регионов со старым поколением. Регионы с молодым поколением делятся на Eden и Survivor. Они осуществляют точно такую же функцию, как и в сборщике мусора Serial GC.

Приложение всегда создаёт объекты в молодом поколении, в Eden, кроме очень больших объектов, которые сразу создаются в старом поколении.

Паузы для сборки мусора G1 очищают место в молодых поколениях полностью, и в дополнение могут очищать место в некотором количестве старых поколениях во время любой из пауз. Во время паузы G1 копирует объекты из выбранных регионов в другие регионы в куче. Регион, куда происходит копирование, зависит от исходного региона: молодое поколение целиком копируется в survivor и старые регионы, а объекты из старых регионов в другие старые регионы.

Алгоритм работы G1 GC:

1. Сборка только в молодом поколении (Young-only phase): Этот этап перетаскивает объекты из регионов с молодым поколением в регионы со старым поколением. Переход между сборкой только в молодом поколении и этапом освобождения памяти происходит в момент достижения определённого порога заполнения кучи, в который G1 стартует процесс пометки только молодого поколения:
   - Начальная пометка (Initial Mark): Помечаются достижимые (живые) объекты в регионах со старым поколением. Этот процесс выполняется БЕЗ stop-the-world. Во время этого процесса также могут происходить сборки только в молодом поколении. После процесса начальной пометки выполняются два этапа с полной остановкой приложения (stop-the-world).
   - Повторная пометка (Remark): Завершает пометку, осуществляет выгрузку классов и глобальную обработку ссылок. Между повторной пометкой и очисткой сборщик мусора вычисляет общую информацию о живых объектах конкурентно (параллельно с работой приложения), которые будут освобождены, эта информация будет использоваться на этапе чистки.
   - Чистка (Cleanup): очищает полностью пустые регионы, определяет нужен ли этап освобождения места.
2. Освобождение места (Space-reclamation phase): Этот этап состоит из сборки мусора, которая в дополнение к молодым регионам также перемещает живые объекты из множества старых регионов. Этап заканчивается, когда G1 решает, что дальнейшее перемещение регионов не приведёт к освобождению достаточного места, чтобы этим заниматься.

JIT-компиляция

Чтобы получить приемлемую скорость исполнения, Java-платформа активно использует динамическую компиляцию в виртуальной Java-машине (JVM). Динамическая (Just-In-Time) компиляция повышает производительность за счет трансляции Java байткода в машинный код в процессе работы приложения. Принцип работы существенно отличается от статических компиляторов и для получения высокопроизводительного кода JVM использует другой набор методик компиляции.

Настоящим компилятором в экосистеме Java является динамический компилятор (JIT). Когда какая-либо инструкция (или набор инструкций) Java-процессора выполняется в первый раз, происходит компиляция соответствующего ей байт-кода с сохранением скомпилированного кода в специальном буфере. При последующем вызове той же инструкции вместо ее интерпретации происходит вызов из буфера скомпилированного кода. Поэтому интерпретация происходит только при первом вызове инструкции.

Сложные оптимизирующие JIT-компиляторы действуют еще изощренней. Поскольку обычно компиляция инструкции идет гораздо дольше по сравнению с интерпретацией этой инструкции, время ее выполнения в первый раз при наличии JIT-компиляции может заметно отличаться в худшую сторону

по сравнению с чистой интерпретацией. Поэтому бывает выгоднее сначала запустить процесс интерпретации, а параллельно ему в фоновом режиме компилировать инструкцию. Только после окончания процесса компиляции при последующих вызовах инструкции будет исполняться ее скомпилированный код. До этого все ее вызовы будут интерпретироваться. Виртуальная машина HotSpot осуществляет JIT-компиляцию только тех участков байт-кода, которые критичны к времени выполнения программы. При этом по ходу работы программы происходит оптимизация скомпилированного кода.

Приложения Java обладают не только хорошей переносимостью, но и высокой скоростью работы. Однако даже при наличии JIT-компиляции они все-таки могут выполняться медленнее, чем программы, написанные на C или C++. Это связано с тем, что JIT-компиляция создает не такой оптимальный код как многопроходный компилятор C/C++, который может тратить очень большое время и много ресурсов на отыскивание конструкций программы, которые можно оптимизировать. А JIT-компиляция происходит "на лету", в условиях жесткой ограниченности времени и ресурсов.

Одним из новшеств Java 9 является Ahead-Of-Time (AOT) компиляция. AOT компиляция – это статическая компиляция и те, кто работал с языком C хорошо знакомы с ней. Суть заключается в том, что на этапе компиляции наш исходный код превращается в исполняемый код. По итогу мы получаем исполняемый файл, который мы можем запустить в любой момент. Соответственно, до начала работы программы нам требуется больше времени. Поэтому данный вид компиляции так и называется Ahead-Of-Time.

Данный вид компиляции имеет следующие основные преимущества:

- Время запуска приложения. Для холодного старта (первого запуска нашей программы) нам необходимо значительно большее время по сравнению с JIT компиляцией. Но, после того, как код статически скомпилирован, время старта не отличается от времени старта такой же программы на языке C.
- Невозможность декомпиляции. Исполняемый код, который мы получаем в результате AOT компиляции можно дизассемблировать со значительными сложностями, а декомпилировать байт код в Java код не составляет трудности.

Литература и ссылки

1. Спецификация языка Java https://docs.oracle.com/javase/specs/jls/se8/html/index.html
2. Шилдт Г. Java 8. Полное руководство. 9-е издание. — М.: Вильямс, 2012. — 1377 с.
3. Эккель Б. Философия Java. 4-е полное изд. — СПб.: Питер, 2015. — 1168 с.

4. Эванс Бенджамин, Вербург Мартин. Java. Новое поколение разработки, 2014.
5. https://habrahabr.ru/post/269621/
6. https://habrahabr.ru/post/269707/
7. https://habrahabr.ru/post/269863/

Вопросы для самоконтроля

1. Назовите основные компоненты Java-машины
2. Назовите сферу применения каждого сборщика мусора в Java
3. Назовите основные JVM-оптимизации